Informatique (Semestre 1)

CM 7
Récursivité

La Prépa, Grenoble INP, UGA

2023-2024

Sommaire

Sommaire

Fonction récursive

La conception d'une fonction récursive est assez proche de la *démonstration par récurrence* en mathématiques :

- Au rang 0, le résultat de la fonction est trivial
- Le résultat au rang n est obtenu à partir du résultat au rang $n-1$

```
def f(n):
    if n == 0:
        return ...
    else:
        return ... f(n-1) ...
```

Factorielle

Définition récursive de la factorielle :

- $0! = 1$
- $\forall n \in \mathbb{N}^* : n! = n \times (n-1)!$

Version récursive de la factorielle :

```
def fact(n):
    assert n >= 0
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

$\rightsquigarrow$ Demo avec PythonTutor : `https://goo.gl/EULC5w`

Fonction puissance

Définition récursive de la fonction puissance :

- $x^0 = 1$
- $\forall n \in \mathbb{N}^*,\ x^n = x \,.\, x^{n-1}$

Code de la fonction puissance :

```
def puissance(x, n):
    if n == 0:
        return 1
    else:
        return x * puissance(x, n - 1)
```

Complexité de la fonction puissance

- On cherche à calculer *l'ordre de grandeur* du temps d'exécution de la fonction.
- On considère que les opérations arithmétiques se font en temps constant (c'est une approximation).
- On note $C(x, n)$ la complexité en temps d'exécution de l'appel `puissance(x, n)`.

On compte :

- comparaison `n == 0` : temps k_C
- opération `-` : temps k_S
- opération `*` : temps k_M

On a :

- $C(x, 0) = k_C$ (1 comparaison)
- $\forall n \in \mathbb{N}^*,\ C(x, n) = C(x, n-1) + k_C + k_S + k_M$
  (l'appel récursif, plus une comparaison, une soustraction et une multiplication).

Complexité de la fonction puissance

On a donc :

$$\begin{array}{lcl}
C(x,n) & = & C(x,n-1) + k_C + k_S + k_M \\
C(x,n-1) & = & C(x,n-2) + k_C + k_S + k_M \\
C(x,n-2) & = & C(x,n-3) + k_C + k_S + k_M \\
\quad \ldots & & \\
C(x,1) & = & C(x,0) + k_C + k_S + k_M \\
C(x,0) & = & k_C \\
\hline
C(x,n) & = & k_C + n\,.\,(k_C + k_S + k_M)
\end{array}$$

Donc $C(x,n) = k_C + n\,.\,(k_C + k_S + k_M) = O(n)$

Le temps d'exécution de la fonction puissance est donc proportionnel à n (complexité linéaire en n).

Puissance rapide

Mais faut-il vraiment 8 multiplications pour calculer x^8 ?
Non, 3 suffisent :

```
x2 = x * x
x4 = x2 * x2
x8 = x4 * x4
```

Comment exploiter cette optimisation pour calculer de manière rapide `puissance(x, n)` ?

On remarque que

- pour $n = 2k$ (n pair), $x^n = x^{2k} = (x^k)^2$;
- pour $n = 2k + 1$ (n impair), $x^n = x^{2k+1} = x \,.\, (x^k)^2$.

Puissance rapide : implémentation

Rappels :

- pour tester si n est pair : `if n % 2 == 0:`
- division euclidienne : `k = n // 2`

Puissance rapide : implémentation

Rappels :

- pour tester si n est pair : `if n % 2 == 0:`
- division euclidienne : `k = n // 2`

```
def puissance_rapide(x, n):
    assert n >= 0
    if n == 0:
        return 1
    else:
        r = puissance_rapide(x, n // 2)
        if n % 2 == 0:
            return r * r
        else:
            return x * r * r
```

Puissance rapide : implémentation

Rappels :

- pour tester si n est pair : `if n % 2 == 0:`
- division euclidienne : `k = n // 2`

```
def puissance_rapide(x, n):
    assert n >= 0
    if n == 0:
        return 1
    else:
        r = puissance_rapide(x, n // 2)
        if n % 2 == 0:
            return r * r
        else:
            return x * r * r
```

Ici encore, n est strictement décroissant et positif à chaque appel récursif : c'est une condition suffisante pour montrer la terminaison.

Puissance rapide : complexité

Comme précédemment, on suppose que les opérations arithmétiques se font en temps constant :

- Condition : temps k_C
- Division : temps k_D
- Reste : temps k_R
- Multiplication : temps k_M

On note $C(x, n)$ la complexité en temps de calcul d'un appel à `puissance_rapide(x, n)`

Puissance rapide : complexité

Dans le meilleur des cas, $n = 2^p$ (donc $p = \log_2(n)$) et la condition `n % 2 == 0` est vraie à chaque appel récursif (sauf pour $p = 0$).

On a alors

$$\begin{array}{rrcl} & C(x,0) & = & k_C \\ & C(x,n) & = & 2k_C + k_D + k_R + k_M + C(x, n/2) \\ \Rightarrow & C(x,2^p) & = & 2k_C + k_D + k_R + k_M + C(x, 2^{p-1}) \end{array}$$

On a donc

$$\begin{array}{lcl} C(x,2^p) & = & 2k_C + k_D + k_R + k_M + C(x, 2^{p-1}) \\ C(x,2^{p-1}) & = & 2k_C + k_D + k_R + k_M + C(x, 2^{p-2}) \\ & \dots & \\ C(x,2^1) & = & 2k_C + k_D + k_R + k_M + C(x, 2^0) \\ C(x,2^0) & = & 2k_C + k_D + k_R + 2k_M + C(x, 0) \\ \hline C(x,2^p) & = & p\,.\,(2k_C + k_D + k_R + k_M) + (3k_C + k_D + k_R + 2k_M) \\ & = & O(p) \\ & = & O(\log_2(n)) \end{array}$$

Puissance rapide : complexité

Dans le pire des cas, $n = 2^p - 1$ et la condition `n % 2 == 0` est fausse à chaque appel récursif.

On a alors

$$
\begin{aligned}
& C(x,0) = k_C \\
& C(x,n) = C(x, 2^p - 1) = 2k_C + k_D + k_R + 2k_M + C(x, 2^{p-1} - 1) \\
\Rightarrow \quad & C(x, 2^p - 1) = k_C + p\,.\,(2k_C + k_D + k_R + 2k_M) \\
\Rightarrow \quad & C(x,n) = k_C + \log_2(n+1)\,.\,(2k_C + k_D + k_R + 2k_M) \\
\Rightarrow \quad & C(x,n) = O(\log_2(n+1)) = O(\log_2(n))
\end{aligned}
$$

On a donc une complexité logarithmique en n dans le meilleur des cas, ainsi que dans le pire des cas. Par conséquent, la complexité *en moyenne* est également logarithmique en n.

L'algorithme de puissance rapide est donc beaucoup plus efficace que l'algorithme de puissance initial, dont la complexité était linéaire.

Méthode de Héron pour le calcul de la racine carrée

On considère la suite $(u_n)_{n\in\mathbb{N}}$, qui calcule une approximation de $\sqrt{x}$:

$$u_0 = x \qquad\qquad u_{n+1} = \frac{u_n + \frac{x}{u_n}}{2}$$

On peut écrire directement la formule en Python :

```
# Version récursive naïve
def heron(x, n):
    if n == 0:
        return x
    else:
        return (heron(x, n-1) + x/heron(x, n-1))/2

print("heron(2, 3) = ", heron(2, 3))
print("heron(2, 23) = ", heron(2, 23))
```

Le programme met plusieurs secondes pour calculer le 23e terme de la suite, pourquoi ?

Héron non efficace : pourquoi ?

```
def heron(x, n):
    if n == 0:
        return x
    else:
        return (heron(x, n-1) + x/heron(x, n-1))/2
```

- `heron(x, n)` appelle 2 fois `heron(x, n-1)`,
- 4 fois `heron(x, n-2)`
- 8 fois `heron(x, n-3)`
- ...
- 2^n fois `heron(x, 0)`

La complexité de ce programme est donc exponentielle.

Une idée pour résoudre ce problème ?

Héron efficace !

Eviter de faire plusieurs fois le même calcul, c'est un principe essentiel en algorithmique (pas seulement en programmation récursive).

Solution : mettre `heron(x, n - 1)` dans une variable locale

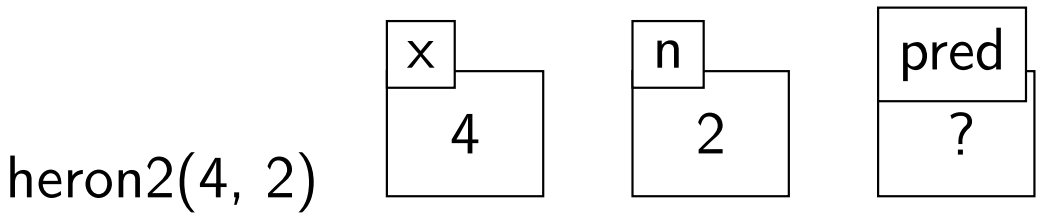
```
def heron2(x, n):
    if n == 0:
        return x
    else:
        pred = heron2(x, n - 1)
        return (pred + x / pred) / 2
print("heron2(2, 3) = ", heron2(2, 3))
print("heron2(2, 23) = ", heron2(2, 23))
```

Le résultat est immédiat !

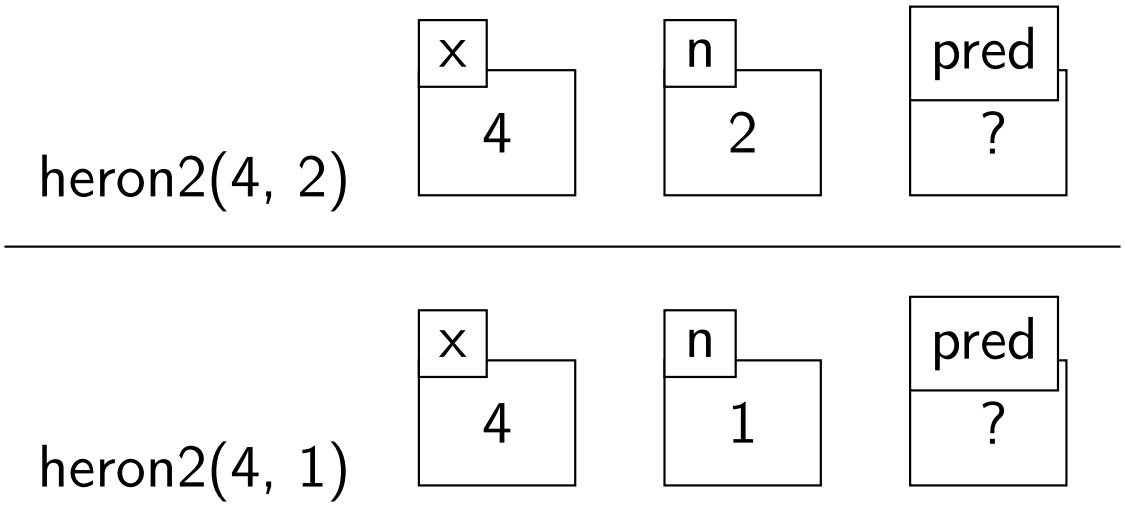
Note : `pred` est une variable locale à la fonction `heron2`, et chaque appel imbriqué de `heron2` définit une nouvelle variable `pred`.

Déroulé de heron2(4, 2)

Au premier appel, on définit la variable `pred`, mais elle n'a pas encore de valeur lorsque `heron2(4, 1)` est appelé :

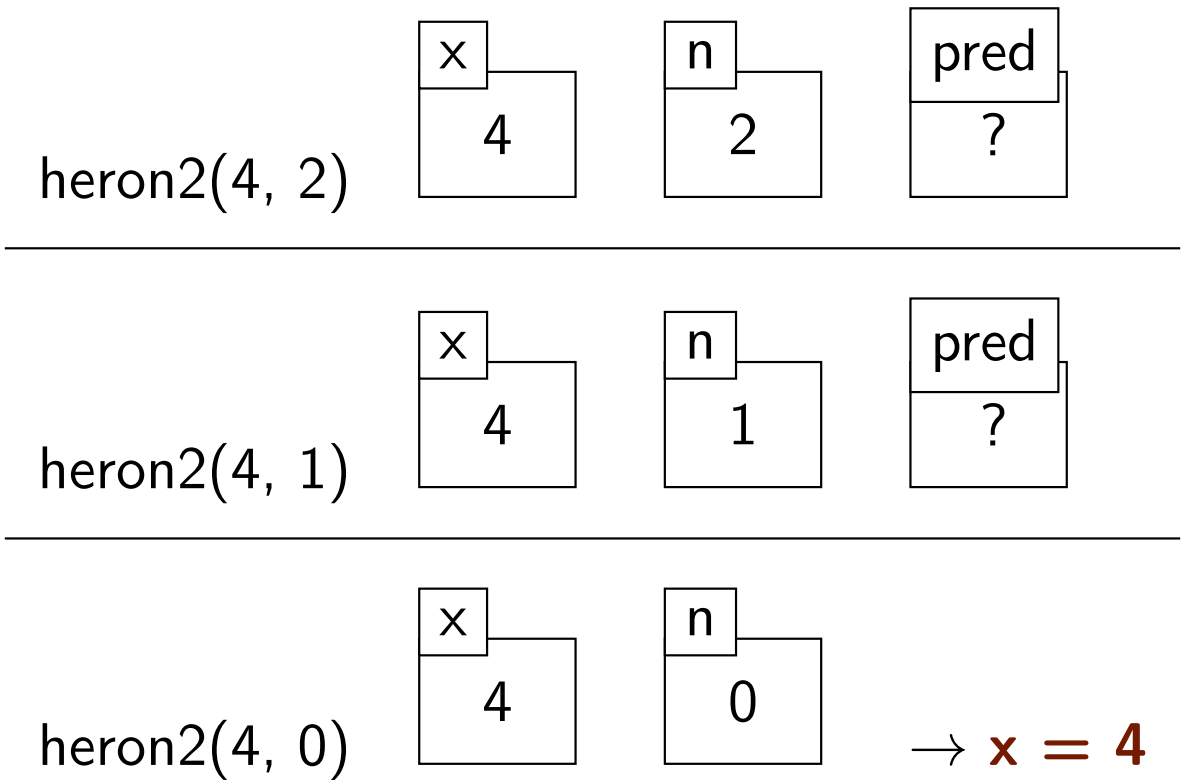


puis on exécute l'appel à heron2(4, 1) :



Déroulé de heron2(4, 2)

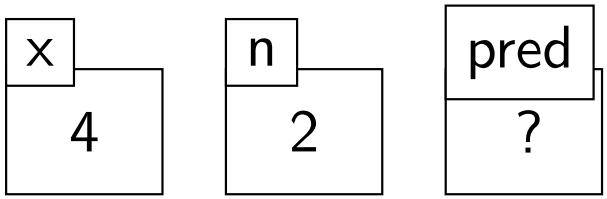
Puis l'appel à `heron2(4, 0)`, qui ne définit pas `pred` mais retourne la valeur 4 :



Déroulé de heron2(4, 2)

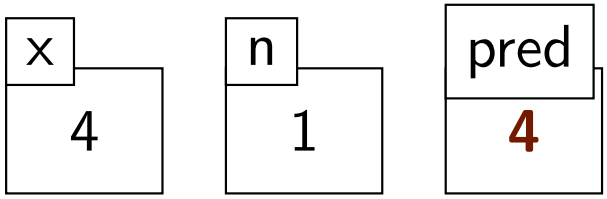
Les variables locales sont alors désallouées, et la variable `pred` de l'appel précédent prend la valeur de retour :

heron2(4, 2)



---

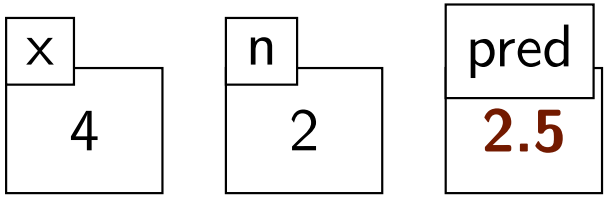
heron2(4, 1)



$\rightarrow$ **(4 + x / 4) / 2 = 2.5**

Et enfin :

heron2(4, 2)



$\rightarrow$ **(2.5 + x / 2.5) / 2 = 2.05**

qui retourne 2.05

La récursivité : avantages / inconvénients

- + ressemble à la récurrence en maths
- + parfois plus naturelle
- + code plus compact
- − mise au point / compréhension du programme plus compliquée
- − calcul de coût plus complexe

Est-elle vraiment nécessaire ? Bien réfléchir selon le cas !

- Factorielle ?
- Puissance ?
- Héron ?

Comme pour les boucles, il est nécessaire de prouver :

- la *correction* (le plus souvent une démonstration par récurrence)
- la *terminaison* (nombre d'appels imbriqués fini) des fonctions récursives

(voir §5.3.3, dans les lectures à faire avant l'exam)

Sommaire

Sommaire de cette section

Diviser pour régner

Principe *diviser pour régner* : pour résoudre un problème, on le divise en problèmes plus petits et plus faciles à résoudre, et on combine les résultats de ces sous-problèmes.

Diviser pour régner récursif pour trier un tableau

- Pour trier 1 tableau de taille n :
  - on divise le tableau en 2 tableaux de taille $n/2$ ...
  - on trie chaque partie :
    - ★ on découpe chaque partie en 2 tableaux de taille $n/4$
    - ★ on trie chaque partie
    - ★ on assemble les tableaux triés
  - on assemble les deux tableaux triés.

Principe du tri fusion

Voyons comment ça fonctionne sur un exemple concret : j'ai 62 copies à trier par ordre alphabétique.

- je sépare le paquet en deux paquets de 31
- je donne 31 copies chacun à deux élèves du premier rang
- je leur demande de les trier
- je récupère les demi-paquets triés
- je fusionne les paquets
  - je compare les deux copies en haut des deux paquets
  - je choisis la première dans l'ordre alphabétique et la retire du paquet. Je recommence jusqu'à ce qu'il n'y a plus de copies dans un des deux paquets.
- le coût de la fusion est d'au plus 61 comparaisons.

Principe du tri fusion

chacun des deux élèves, pour trier son paquet :

- sépare le paquet en deux paquets de 15 et 16 copies
- donne chacun des deux paquets à deux élèves derrière lui pour qu'ils les trient
- récupère les demi-paquets triés
- les fusionne
- rend le paquet de 31 copies trié à l'élève qui lui avait confié.

Et ainsi de suite. Un élève qui reçoit 1 ou 2 copies à trier les trie lui-même et repasse tout de suite devant.

Intérêt du « diviser pour régner » : intuition

- On part d'un algorithme de tri en n^2
- 1^{er} niveau : 2 tableaux de $\frac{n}{2}$ à trier $\Rightarrow 2 \cdot \left(\frac{n}{2}\right)^2 = \frac{n^2}{2}$
- 2^{eme} niveau : 4 tableaux de $\frac{n}{4}$ à trier $\Rightarrow 4 \cdot \left(\frac{n}{4}\right)^2 = \frac{n^2}{4}$
- Trier un tableau 2 fois plus petit fait moins que 2 fois moins de travail $\Rightarrow$ c'est rentable de découper !

Complexité en nombre de comparaisons pour n copies

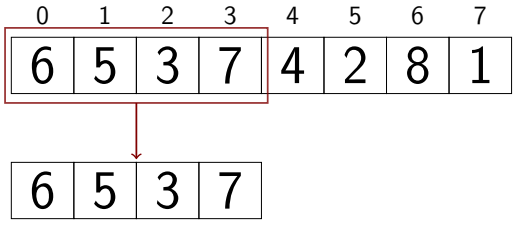
- si n, était une puissance de deux, chacun aurait au rang k exactement $n/2^k$ copies à traiter, donc le rang maximum (celui où tout le monde a exactement deux copies à trier) est $k = \log_2(n) - 1$
- si n n'est pas une puissance de deux, on obtient un majorant de k en majorant n par la puissance de 2 supérieure. On obtient $k = \lceil \log_2(n) \rceil - 1$. Pour $n = 62$, $k = 5$
- chaque élève, quand il commence son travail, a très peu de boulot au début (séparation), ensuite il se tourne les pouces en attendant le résultat, puis il a un peu de boulot à la fin (fusion). Au total, pour faire la fusion, il a moins de m comparaisons à faire, où m est la somme des tailles des paquets qu'il a reçu.
- chaque rang de la classe "traite" au total exactement $n = 62$ copies (y compris moi, qui suis au rang 0), il y a donc moins de n comparaisons à faire à chaque rang.
- Donc, complexité totale en nombre de comparaisons est inférieure à $n(\lceil \log_2(n) \rceil - 1) = O(n \log_2(n))$

Exemple

0	1	2	3	4	5	6	7
6	5	3	7	4	2	8	1

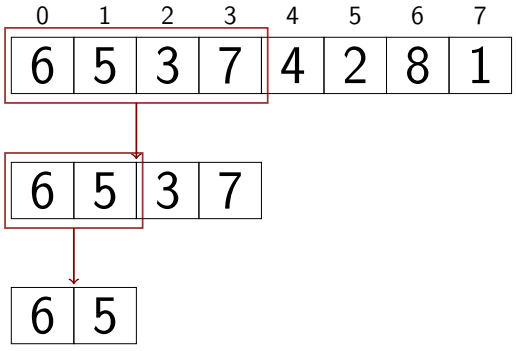
Partage : trier `a` $\rightsquigarrow$ trier `a[0:4]` et `a[4:8]`

Exemple



Partage : trier `a` $\rightsquigarrow$ trier `a[0:4]` et `a[4:8]`

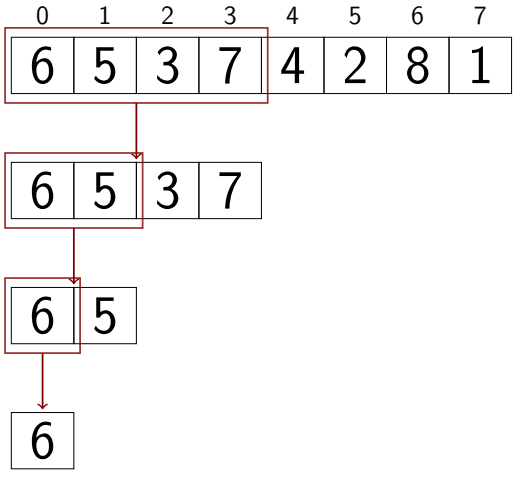
Exemple



Partage : trier `a` $\rightsquigarrow$ trier `a[0:4]` et `a[4:8]`

Partage : trier `a[0:4]` $\rightsquigarrow$ trier `a[0:2]` et `a[2:4]`

Exemple

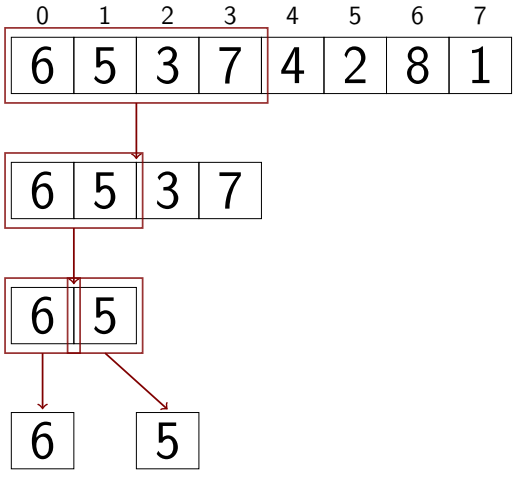


Partage : trier `a` $\rightsquigarrow$ trier `a[0:4]` et `a[4:8]`

Partage : trier `a[0:4]` $\rightsquigarrow$ trier `a[0:2]` et `a[2:4]`

Partage : trier `a[0:2]` $\rightsquigarrow$ trier `a[0:1]` et `a[1:2]`

Exemple

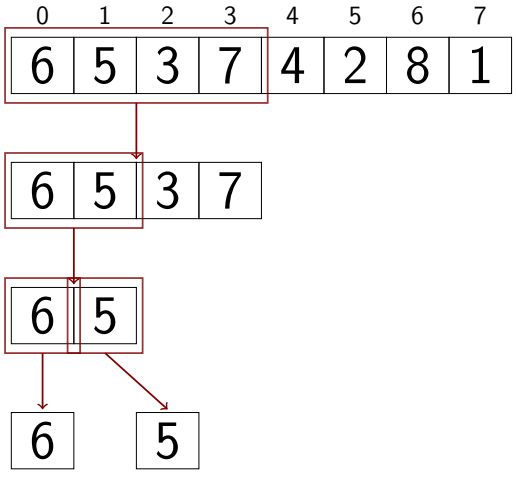


Partage : trier `a` $\rightsquigarrow$ trier `a[0:4]` et `a[4:8]`

Partage : trier `a[0:4]` $\rightsquigarrow$ trier `a[0:2]` et `a[2:4]`

Partage : trier `a[0:2]` $\rightsquigarrow$ trier `a[0:1]` et `a[1:2]`

Exemple



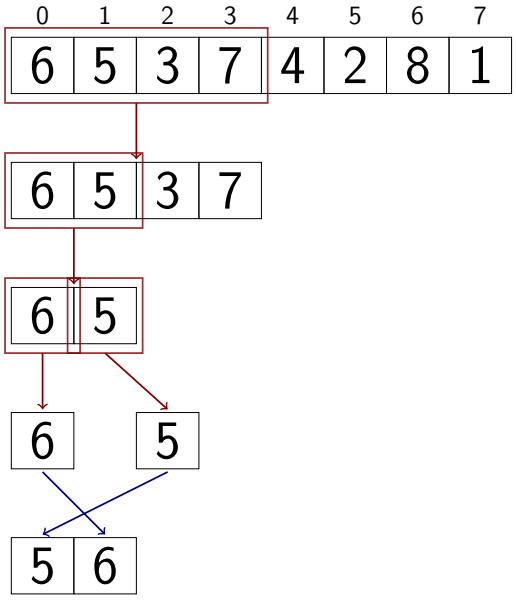
Partage : trier `a` $\rightsquigarrow$ trier `a[0:4]` et `a[4:8]`

Partage : trier `a[0:4]` $\rightsquigarrow$ trier `a[0:2]` et `a[2:4]`

Partage : trier `a[0:2]` $\rightsquigarrow$ trier `a[0:1]` et `a[1:2]`

Listes de taille 1 : rien à faire !

Exemple



Partage : trier `a` $\rightsquigarrow$ trier `a[0:4]` et `a[4:8]`

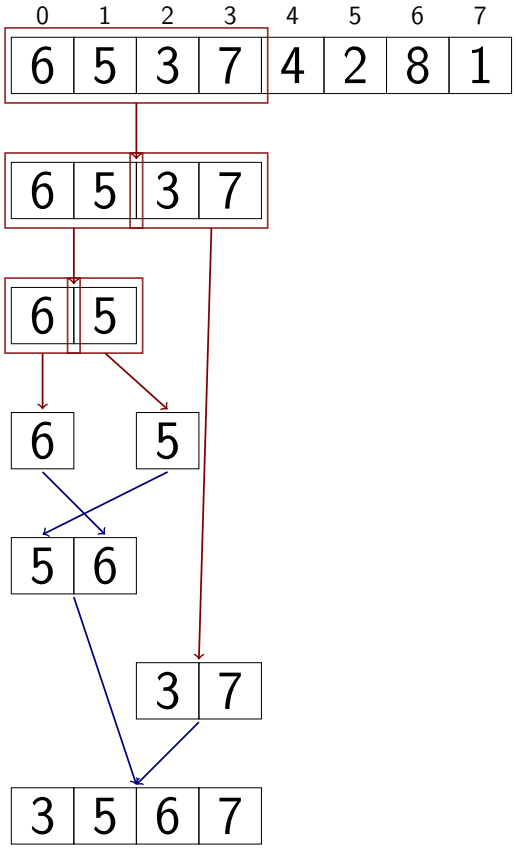
Partage : trier `a[0:4]` $\rightsquigarrow$ trier `a[0:2]` et `a[2:4]`

Partage : trier `a[0:2]` $\rightsquigarrow$ trier `a[0:1]` et `a[1:2]`

Listes de taille 1 : rien à faire !

Fusion

Exemple



Partage : trier `a` $\rightsquigarrow$ trier `a[0:4]` et `a[4:8]`

Partage : trier `a[0:4]` $\rightsquigarrow$ trier `a[0:2]` et `a[2:4]`

Partage : trier `a[0:2]` $\rightsquigarrow$ trier `a[0:1]` et `a[1:2]`

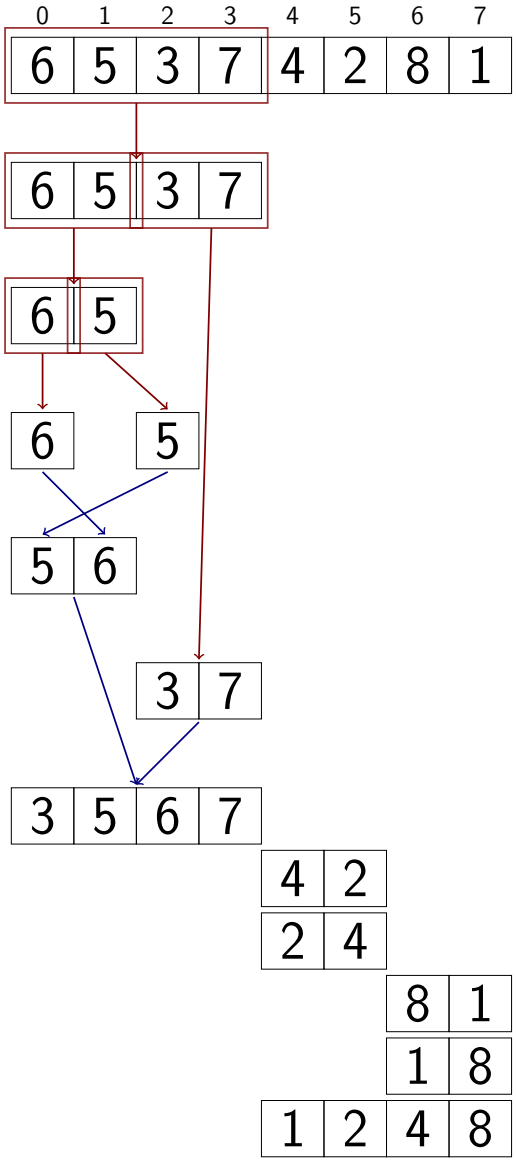
Listes de taille 1 : rien à faire !

Fusion

Partage puis Fusion

Fusion

Exemple



Partage : trier `a` ⇝ trier `a[0:4]` et `a[4:8]`

Partage : trier `a[0:4]` ⇝ trier `a[0:2]` et `a[2:4]`

Partage : trier `a[0:2]` ⇝ trier `a[0:1]` et `a[1:2]`

Listes de taille 1 : rien à faire !

Fusion

Partage puis Fusion

Fusion

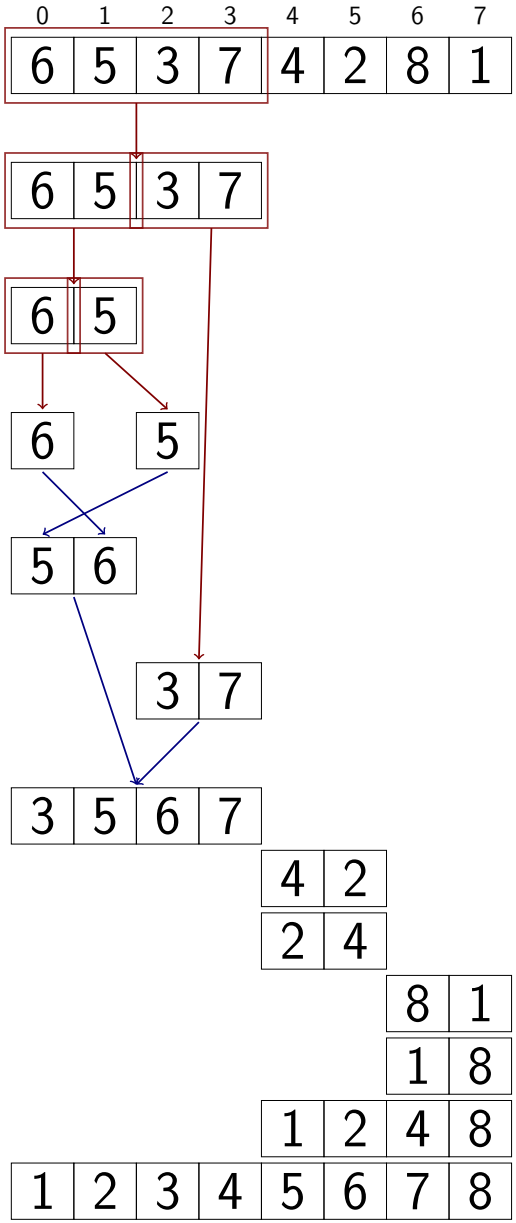
Partage : trier `a[4:6]` ⇝ trier `a[4:5]` et `a[5:6]`

Fusion

Partage : trier `a[6:8]` ⇝ trier `a[6:7]` et `a[7:8]`

Fusion

Exemple



Partage : trier `a` $\rightsquigarrow$ trier `a[0:4]` et `a[4:8]`

Partage : trier `a[0:4]` $\rightsquigarrow$ trier `a[0:2]` et `a[2:4]`

Partage : trier `a[0:2]` $\rightsquigarrow$ trier `a[0:1]` et `a[1:2]`

Listes de taille 1 : rien à faire !

Fusion

Partage puis Fusion

Fusion

Partage : trier `a[4:6]` $\rightsquigarrow$ trier `a[4:5]` et `a[5:6]`

Fusion

Partage : trier `a[6:8]` $\rightsquigarrow$ trier `a[6:7]` et `a[7:8]`

Fusion

Fusion

Sommaire de cette section

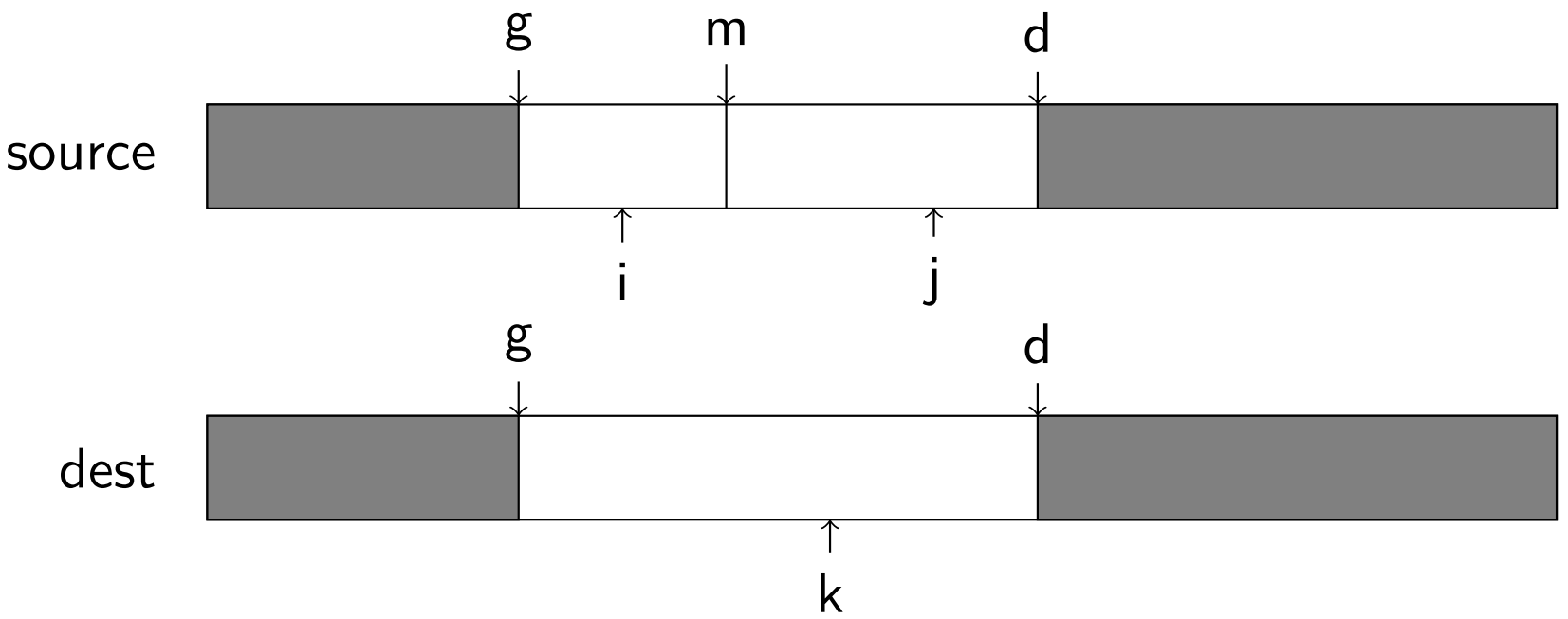
Codage du tri fusion en Python : préliminaire

- On a besoin de coder le tri d'une partie d'un tableau `a`, des indices `g` (inclus) à `d` (exclu).
- Pour le partage, il suffit de calculer l'indice médian $m = (g + d)/2$
- La fusion vraiment « en place » est très compliquée. On simplifie le problème en utilisant un tableau auxiliaire : pour fusionner `a[g:m]` et `a[m:d]` :
  - recopie de `a[g:d]` vers `tmp[g:d]`
  - fusion de `tmp[g:m]` et `tmp[m:d]` vers `a[g:d]`
- Pour la fusion, les zones à fusionner sont toujours contiguës. Il faut donc 3 indices : 1 dans chaque zone à fusionner du tableau source (`tmp[g:m]` et `tmp[m:d]`), et 1 dans le tableau destination.

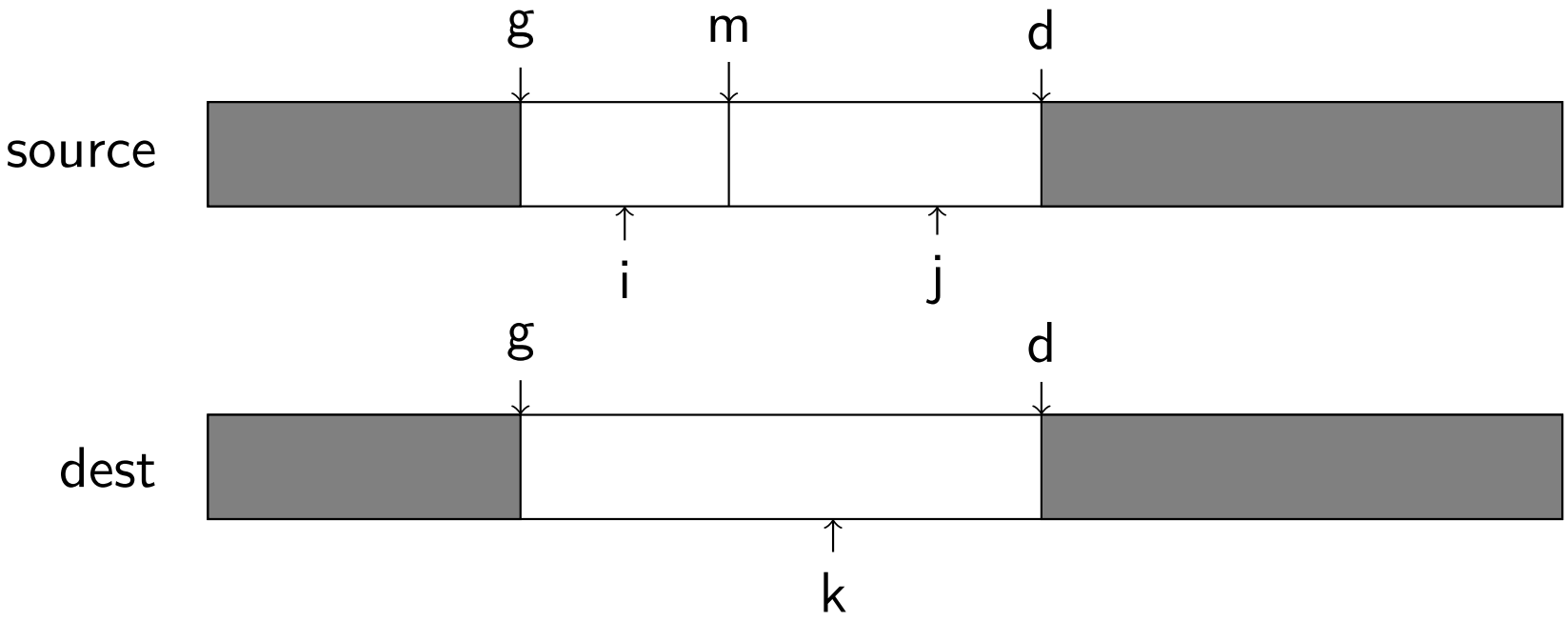
(Note : à partir d'ici, on regarde la fonction "fusion" donc on prend des noms de variables plus expressifs dans ce contexte : `source` et `dest` au lieu de `tmp` et `a`)

Fusion de listes

Écrivons la fonction fusion, qui fusionne les portions `source[g:m]` et `source[m:d]` dans la portion `dest[g:d]`. Pour cela, il faut parcourir les deux portions avec deux indices distincts, et quand on "prend" un élément dans une portion, on avance l'indice correspondant.



Fusion de listes



L'itération consiste donc :

- à déterminer lequel des éléments `source[i]` ou `source[j]` doit être sélectionné, en faisant bien attention au cas où il n'y a plus d'élément à prendre dans une des moitiés
- à copier cet élément dans le tableau `dest`
- à incrémenter i ou j
- à incrémenter k

```
def fusion(source, dest, g, m, d):
    """Fusionne les sous-tableaux source[g:m] et
    source[m:d] dans le tableau dest.

    g, m, d: gauche, milieu, droite.

    dest doit être assez grand."""
    i = g
    j = m
    for k in range(g, d):
        if i < m and (j == d or
                      source[i] <= source[j]):
            dest[k] = source[i]
            i = i + 1
        else:
            dest[k] = source[j]
            j = j + 1
```

Tri fusion et récursivité

- Le tri fusion lui-même est une procédure récursive, puisque pour faire le tri il faut d'abord faire le tri des deux sous-parties.
- Reste à écrire : fonction (récursive) qui trie la section $[g, d[$ du tableau `a`, en utilisant un tableau `tmp` de même taille.

```
def tri_fusion_rec(a, tmp, g, d):
    """Trie le tableau a entre les indices
    g (inclu) et d (exclu).

    Utilise le tableau tmp comme tableau auxiliaire
    (tmp doit être de la même taille que a)."""
    # si l'intervalle fait un seul élément...
    if g + 1 >= d:
        return  # ... il n'y a rien à faire.
    m = (g + d) // 2  # m = milieu de liste
    tri_fusion_rec(a, tmp, g, m)
    tri_fusion_rec(a, tmp, m, d)
    # recopie dans le tableau auxiliaire
    for i in range(g, d):
        tmp[i] = a[i]
    fusion(tmp, a, g, m, d)
```

Tri fusion : fonction pour l'utilisateur

Et pour l'utilisateur, il faut écrire une fonction "chapeau" avec des paramètres plus simples :

```
def tri_fusion(a):
    tmp = list(a) # copie de a dans tmp
    tri_fusion_rec(a, tmp, 0, len(a))

t = [6, 5, 3, 7, 4, 2, 8, 1]
print("avant :", t)
# Affiche avant : [6, 5, 3, 7, 4, 2, 8, 1]
tri_fusion(t)
print("après :", t)
# Affiche après : [1, 2, 3, 4, 5, 6, 7, 8]
```

Sommaire

Avant l'examen

- Exécuter sur PythonTutor `heron` et `heron2`. Le code du tri fusion est sur la page web du cours.
- Comme d'habitude, le QCM (QCM 7) !
- L'examen de l'an dernier avec une correction est disponible sur Chamilo.

L'examen

- L'examen final du semestre 1 (durée de 2h00) a lieu le mercredi 29 novembre.
- Aucun document n'est autorisé pendant l'examen. Les calculatrices sont également interdites. L'aide mémoire Python déjà distribué sera fourni en annexe du sujet.